A Genetic Algorithm for Approximating Relative Maxima

MAT362 final project

Michael Middleton

**A Genetic Algorithm for Approximating Relative Maxima**

# Abstract

Genetic algorithms, while complex and diverse offer a unique approach to modeling a scenario and optimizing a specific outcome based on a fitness function. The core mechanic of a genetic algorithm involves taking a randomly generated population of “chromosomes” modeled by a binary sequence of a certain length and breeding a new population of chromosomes from the most-fit chromosomes chosen to be parents based on some fitness function. The most-fit chromosomes have more bits in the correct genetic position than those determined to be less-fit.

The bits themselves represent a decimal value which when entered as an input to the fitness function will produce an output along the function curve. The best-fit chromosome would be a binary sequence which evaluates to a decimal input into the fitness function which would yield an output on the apex of the available input. Each successive generation of chromosomes bred from the most-fit chromosomes from the previous generation will result in an input into the fitness function producing an output which is closer to the optimal chromosome.

The purpose of this project as I approached it was to implement a genetic algorithm and all of its components which would simulate this population change over time and chart its convergence to the most optimal chromosome.

# Project

To manage the many complexities that a program that this scale presents, I abstracted away the immediate complexity of much of the algorithm’s operation to helper functions which perform various helper tasks that the algorithm uses. In essence, the stages of the algorithm can be seen below in Figure1:

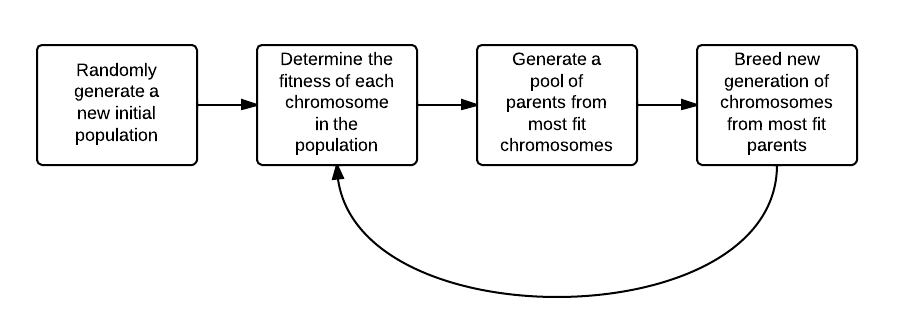


Figure 1: A depiction of the algorithm

The scenario I chose in which to apply the genetic algorithm is of relative mathematic simplicity, yet complex enough to demonstrate the functionality of the code. My fitness function relates the “coolness” of an individual with respect to beard length where beard length is represented by a genetic chromosome sequence.

The genetic algorithm script in the file Genetic\_Algorithm.m shown in Appendix A begins by declaring the length of a genetic sequence to be 10 bits and a population to be 15 chromosomes. We then create a randomly generated initial population by calling the new\_initial\_population function. Then with the use of a library map and my fitness function (and helper functions), I determined the fitness of each chromosome and associated that chromosome with its fitness value by storing them together in the library map.

Once each chromosome is associated with its fitness value, the chromosomes are sorted and the 6 most-fit chromosomes from the current generation are selected to be parents; this process is done in the find\_most\_fit function. Then, the breed function is used to create a new generation of chromosomes using a 2-point crossover method which is a common breeding technique for many genetic algorithms. The 2-point crossover method can be seen below in Figure 2.

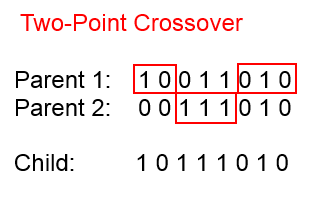


Figure 2: 2-point crossover breeding method

The two-point crossover breeding method randomly chooses 2 split-points, one on each half of the genetic sequence and divides 2 randomly selected parents along those split points. The child inherits everything below the first split point from parent 1, the sequence between the two split points from parent 2, and then the remaining sequence following split point 2 from parent 1. The sequence is repeating until the pool of the 6 most fit parents have bred an entire new population.

The process is then repeated for the designated number of generations. At the end of the desired number of generations, the single most-fit chromosome is plotted, showing the most-fit approximation yielded from the genetic algorithm.

# Analysis

I ran my algorithm over a course of 100 populations, where each new generation would ideally pass the fitness of the previous generations yielding an even more fit new population. As shown below in Figure 3, the final most-fit chromosome of the final generation is marked with a red asterisk.

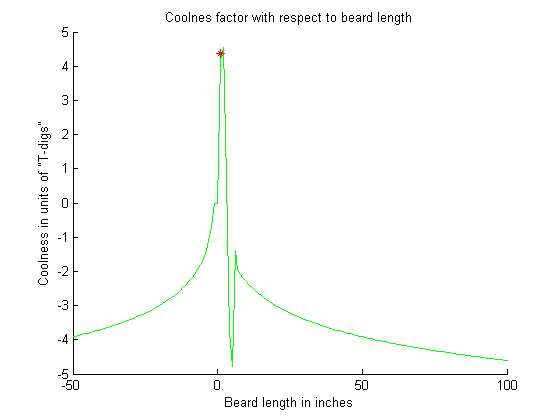


Figure 3: Most-fit chromosome on fitness function

As can be seen above, the most-fit chromosome from the final generation is a very close approximation to the relative maxima of the interval over which the beard lengths were designated. In terms of the most cool person for its given beard length, we can see that the maximum “coolness” comes from a beard of about 4-5 inches.

# Conclusion

Realistically, genetic algorithms can get rather complicated very quickly depending on the scenario which is being simulated. This algorithm alone has 3 nested for loops giving it a O(n^3) complexity. If finding the input of a function which will give us the maxima of a function is all that is necessary, there are plenty of more optimized solutions that will both converge more quickly and with better big-O complexity.

Where genetic algorithms exceed is in the information they provide during the scenarios themselves. A genetic algorithm simulates the progression of a “survival-of-the-fittest” scenario over a given number of population generations, which open up a whole new realm of simulation possibilities. Applications in gene sequencing, population modeling, control systems and any number of disciplines can utilize genetic algorithms to present more data about the specifics of each element, as they approach a fitness function’s relative maxima.

# Appendix A

% -----------------------------------------------------------------------

% Genetic\_Algorithm.m

% Author: Michael Middleton

% Version: 0.1

% Description: This script contains a collection of functions which will

% be used in the implementation of a genetic algorithm which will simulate

% the genetic alterations of a genetic sequence of triats over multiple

% populations of a specific species

% ----------------------------------------------------------------------

hold on

gene\_length = 10;

pop\_size = 15;

population = new\_initial\_population(gene\_length, pop\_size);

champions = [];

% track over 100 generations

for i = 1:100

% create a new library map

gen\_fit = containers.Map

for j = 1:size(population,1)

% associate the fitness of a chromosome with the chromosome itself

gen\_fit(num2str(j)) = cool(binary\_to\_decimal(population(j,:)));

end

vals = cell2mat(values(gen\_fit));

% find 6 most fit parents in the population

fitnesses = sort(vals,'descend');

breeders = zeros(6,gene\_length);

chromosome\_keys = keys(gen\_fit);

for k = 1:6

coolio = fitnesses(k);

for l = 1:length(chromosome\_keys)

temp = chromosome\_keys(l);

temp2 = char(temp);

% if found a cool key, add to breeders and remove from keys

if gen\_fit(temp2) == coolio

breeders = [breeders; population(str2num(temp2),:)];

%remove(gen\_fit,chromosome\_keys(l));

break

end

end

end

champions = [champions; find\_most\_fit(population)];

% breed a new population of chromosomes from most fit parents

population = breed\_new\_generation(breeders,pop\_size,gene\_length);

end

x = [-50:100];

y = zeros(1,length(x));

for i = 1:length(x)

y(i) = cool(x(i));

end

plot(x,y,'g')

plot(-5\*sin(max(champions)), '\*r')

% / ---------------------------------------------------------------------

% / cool():

% / This is the fitness function. It depicts the coolness of a person

% / given their beard length. This function was developed by trial and

% / error until a clear maxima was visable over a given interval

% / ---------------------------------------------------------------------

function coolness = cool(chromosome)

if ((chromosome >= 0) && (chromosome <= 2\*pi))

coolness = 5\*sin(chromosome);

else

coolness = -log(chromosome);

end

end

% -----------------------------------------------------------------------

% find\_most\_fit()

% Description: this function will take in a population and determine the

% single most fit chromosome in that population based on our fitness

% function

% -----------------------------------------------------------------------

function most\_fit\_chromosome = find\_most\_fit(population)

cool\_chromosomes = zeros(1,size(population,1));

for i = 1:size(population,1)

cool\_chromosomes(i) = cool(binary\_to\_decimal(population(i)));

end

most\_fit\_chromosome = max(cool\_chromosomes);

end

% -----------------------------------------------------------------------

% Breed()

% Description: this function will take the genetic sequence of two

% different parents and produce an offspring with a new genetic sequence.

% This function will utilize the mutate() function which will simulate

% the rare occurance of a genetic mutation. The algorithm is based off

% of a 2-point cross-over selection technique, common to genetic

% algorithm reproduction.

% -----------------------------------------------------------------------

function child = breed(parent\_a, parent\_b)

% generate 2 cross-over points

point1 = floor(rand\*ceil((length(parent\_a)/2)));

% accounts for an index error which occurs when point1 is 1

if point1 == 0

point1 = point1 + 1;

end

point2 = ceil(rand\*ceil((length(parent\_a)/2))) + point1;

% produce child from 2 parents

child = [parent\_a(1:point1), parent\_b(point1+1:point2), parent\_a(point2+1:length(parent\_a))];

child = mutate(child);

end

function newborn\_generation = breed\_new\_generation(most\_fit\_parents, population\_size, gene\_length)

% set space for new population

newborn\_generation = zeros(population\_size, gene\_length);

for i = 1:population\_size

% select 2 parents from most fit - assuming 6 most fit

parent\_a = most\_fit\_parents(ceil(rand\*6),:);

parent\_b = most\_fit\_parents(ceil(rand\*6),:);

% breed new child chromosome

child = breed(parent\_a, parent\_b);

newborn\_generation(i,:) = child;

end

end

% -----------------------------------------------------------------------

% binary\_to\_decimal.m

% Author: Michael Middleton

% Version: 0.1

% Description: This function will calculate the decmimal equivalent of

% the binary chromosome genetic sequence

% ----------------------------------------------------------------------

function decimal\_chromosome = binary\_to\_decimal(input\_chromosome)

val = 0;

for i=1:length(input\_chromosome)

if input\_chromosome(i) == 1

val = val + 2^(length(input\_chromosome)-(i));

else

continue

end

end

decimal\_chromosome = val;

end

% -----------------------------------------------------------------------

% Mutate()

% Description: this function will take in a particular genetic sequence

% and simulate the mutation of various characteristics based on a

% set of statistical probabilities.

% -----------------------------------------------------------------------

function mutated\_chromosome = mutate(chromosome)

mutated\_chromosome = chromosome; % copies genetic sequence

for i=1:length(chromosome) % analyze every bit in genetic sequence

prob = rand;

if prob >= .99 % 1/100 chance of a bit mutation

if mutated\_chromosome(i) == 0

mutated\_chromosome(i) = 1;

else

mutated\_chromosome(i) = 0;

end

else

continue

end

end

end

% -----------------------------------------------------------------------

% new\_initial\_population()

% Description: this function will take in the size of the population

% and the length of the genetic sequence each chromosome will have

% and will return a randomly generated population of genetic samples

% -----------------------------------------------------------------------

function population = new\_initial\_population( genetic\_length, population\_size )

% give new population the appropriate dimensions

population = zeros(1,genetic\_length);

for i=1:population\_size+1

chrom = new\_random\_chromosome(genetic\_length);

population = [population; chrom];

end

population = population(2:population\_size+1,:);

end

% -----------------------------------------------------------------------

% new\_random\_chromosome()

% Description: this function will generate a new genetic sequence of

% the length taken as an input parameter. It will then return to the

% invocation the binary sequence

% -----------------------------------------------------------------------

function random\_chromosome = new\_random\_chromosome(chromosome\_length)

random\_chromosome = rand(1,chromosome\_length);

for i=1:chromosome\_length

if random\_chromosome(i) < .5

random\_chromosome(i) = 0;

else

random\_chromosome(i) = 1;

end

end